第3章

智能软件操作系统

3.1 昇腾软件体系架构

昇腾的软件体系是一个典型的、层层递进的全栈式架构，其设计理念是软硬件协同与分层解耦。每一层都构建于下层能力之上，并向上层提供更抽象、更易用的接口与服务，最终目标是将底层专用硬件的强大算力高效、便捷地释放给最终用户和开发者。该体系自底向上可分为四个核心层次，共同构成了从硬件到智能应用的完整桥梁。

整个体系的基石是操作系统层，通常以 Ubuntu 或 openEuler 作为基础平台。这一层不仅负责硬件资源管理、进程调度、文件系统和网络通信等核心服务，更是软件与昇腾硬件首次握手的关键所在——其内核集成的昇腾驱动，如同一位精准的翻译官，将标准的操作系统指令转换为硬件能理解的信号，使得上层软件能够安全、高效地识别并控制昇腾 AI 处理器硬件。

构建在操作系统之上的是整个体系的心脏与大脑——异构计算架构(CANN)。作为连接上层生态与底层硬件的核心桥梁，CANN 是实现软硬件协同的关键。它包含直接管理硬件设备的驱动层、承担上下衔接功能的运行时库、负责芯片内部任务调度的“交通指挥官”，以及作为性能优化引擎的图编译器。其中，图编译器通过算子融合、常量折叠和数据布局转换等深度优化，能将神经网络模型转换为高度优化的离线模型；而 AscendCL 则提供了面向开发者的底层 API，允许进行极致的硬件控制。正是 CANN 对硬件架构的深刻理解，才使得计算图能被翻译成硬件直接执行的高效指令流。这部分将在第 5 章展开描述。

面向广大开发者的是框架与模型层，这一层提供了友好而强大的开发界面。华为原生的昇思(MindSpore)框架与昇腾硬件达到了“血脉相通”的协同级别，支持动静统一的开发模式和自动并行等高级特性。同时，通过“适配器”模式，PyTorch、TensorFlow 等主流框架也能无缝融入昇腾生态，开发者仅需简单修改代码即可将计算任务迁移到昇腾处理器，这种设计有效保护了开发生态和现有投资。这部分将在第 4 章展开描述。

在最顶层的应用与工具层，昇腾体系的价值得到最终实现。行业解决方案将 AI 能力落地到智慧城市、自动驾驶等具体场景，预训练模型库为开发者提供了“开箱即用”的优化模型。而全栈集成开发环境 MindStudio 则是提升开发效率的“神器”，它集成了模型转换、性能分析、应用调试等工具，其性能分析器能够清晰地展

示模型在执行时的算子耗时、数据搬运瓶颈等情况，是软硬件协同思想在开发流程中的集中体现。

从开发者在 Ubuntu 或 openEuler 系统上使用 MindSpore 或 PyTorch 编写模型，到通过 CANN 图编译器转换为高性能离线模型，再到通过 AscendCL 或框架接口在硬件上执行计算，最终通过 MindStudio 进行性能优化并打包成行业解决方案——这一完整的协同工作流，充分体现了昇腾软件体系如何通过各层次的精密配合，将原始的 AI 算力转化为切实的商业价值，为各行各业的智能化转型提供了坚实的技术基础。

3.2 昇腾操作系统

昇腾人工智能计算体系在操作系统层面的战略布局，体现为一种兼容并蓄的双轨制策略，其基础平台同时支持全球流行的 Ubuntu 和面向数字基础设施的 openEuler 这两大 Linux 发行版。这一选择并非简单的技术并列，而是一项深思熟虑的生态战略，旨在通过不同特质的操作系统满足从广泛开发到深度定制的多元化需求，共同为上层复杂的 AI 软件栈构筑起坚实而灵活的运行基石。

Ubuntu 作为源自“乌班图”理念的开源系统，以其卓越的易用性和庞大的社区力量，为昇腾生态提供了强大的大众化入口。该系统基于 Debian 发行版，凭借其深厚的桌面应用基因和对用户体验的极致追求，为开发者在昇腾硬件上进行编程、调试和图形化操作带来了无缝的过渡体验。其庞大的软件仓库和高效的包管理机制，使得 AI 开发所需的各种工具、库和依赖能够被轻松获取与部署。更重要的是，Ubuntu 背后全球性的活跃社区构成了一个巨大的知识库，任何开发者在探索昇腾技术过程中遇到的难题，几乎都能从中找到解决方案，这种强大的社区支持极大地降低了昇腾 AI 技术的入门门槛，使其成为学术研究、快速原型验证和融入全球 AI 开发生态的理想“创新沙盒”与“快速通道”。

与 Ubuntu 形成战略互补的是华为推出的 openEuler 操作系统，这是一个专注于服务器、云计算、边缘计算等企业级场景的开源平台。openEuler 从设计之初就致力于构建一个支持多处理器架构的统一、开放的操作系统社区，其技术内核针对数字基础设施的高性能、高可靠性和高安全性需求进行了深度优化。它在内核调度、内存管理及网络栈等方面的增强，为需要持续稳定运行的 AI 推理服务提供了坚实基础。尤为关键的是，openEuler 对多样性计算的原生支持，使其能够与昇腾 AI 处理器实现深度的软硬件协同，其内核能更高效地管理昇腾这类专用加速器。通过开源社区的合作模式，openEuler 持续推动着软硬件应用生态的繁荣，为昇腾在企业关键场景中的部署提供了所需的可靠性、安全性和长期维护保障，堪称昇腾生态的“生产平台”与“优化引擎”。

综上所述，昇腾采用 Ubuntu 与 openEuler 的双轨操作系统策略，巧妙地平衡了生态广度与技术深度。通过 Ubuntu 的广泛接纳度和强大社区，昇腾有效地降低了技术门槛，吸引了全球开发者，构建起庞大的用户基础；而通过 openEuler 的企业级特性和深度优化，则为关键行业客户提供了高性能、高可靠性的解决方案，实现了软硬件一体化的深度协同。这种兼具开放性与纵深性的布局，使得昇腾 AI 既能“接地气”地融入全球开源主流，又“有底气”地服务于最严苛的数字化核心场景，为其在激烈的市场竞争中构建完整且健康的生态系统奠定了坚实基础。

3.2.1　昇腾操作系统构成及运行流程

一个完整的嵌入式 Linux 系统，在逻辑上可以划分为四个紧密协作的层次：引导加载程序、Linux 内核、文件系统以及用户应用程序。这四个部分共同构成了一个从硬件上电到用户交互的完整软件栈。

系统启动始于引导加载程序(BootLoader)，它是设备通电或复位后第一个运行的软件。其核心使命是执行底层的硬件初始化，例如设置时钟、配置内存控制器和串口，为内核的运行准备好一个基本的硬件环境。随后，BootLoader 的主要任务是从 Flash 或 eMMC 等存储介质中将 Linux 内核的映像文件加载到内存中。在此过程中，一个关键的步骤是将预先设置好的启动参数(如根文件系统的位置、控制台配置等)传递给内核，这如同将一封“说明书”交给了内核，告诉它该如何启动。在常见的嵌入式系统开发中，U-Boot 无疑是最常用且最核心的 BootLoader。作为一款功能强大且完全开源的可移植引导加载程序，它支持包括 ARM、MIPS、PowerPC 和 RISC-V 在内的几乎所有主流嵌入式处理器架构。

当内核被加载并开始执行后，系统便进入了嵌入式 Linux 内核的舞台。作为整个操作系统的核心，内核负责全面接管系统的软硬件资源，包括进程调度、内存管理、中断处理和设备驱动等。它会根据 BootLoader 传递来的参数，进行自身的初始化，并完成一个至关重要的操作——挂载根文件系统。根文件系统是 Linux 系统赖以运行的基础环境，其中包含了操作系统运行所必需的所有程序、命令工具、系统库(如 glibc)和配置文件等。在嵌入式设备中，为了节省资源，通常会使用 BusyBox 来提供一个精简而强大的命令工具集，而文件系统的格式也会根据存储硬件的不同进行选择，如针对 NAND Flash 的 UBIFS 或针对 eMMC 的 EXT4。

在根文件系统成功挂载之后，内核会启动第一个用户空间进程(通常是/sbin/init)，系统启动的接力棒也就交给了最后一个层次——用户应用程序。这些应用程序是最终体现设备功能的软件，可以是实现特定控制逻辑的后台服务、处理网络通信的协议栈，甚至是搭载了图形用户界面(GUI)的交互式程序，它们通过调用内核提供的系统接口和文件系统中的库文件来运行，最终让嵌入式设备实现其设计目的。

综上所述，嵌入式 Linux 系统的启动是一个环环相扣的连续过程：从 BootLoader 的硬件初始化开始，到内核加载与资源管理，再到文件系统挂载提供运行环境，最终由用户应用程序执掌系统，实现完整功能。这一清晰的分层结构与启动流程，奠定了嵌入式 Linux 系统稳定与灵活的基石。整个流程如图 3.1 所示。

3.2.2　昇腾 Linux 操作系统内核

Linux 内核作为操作系统的核心，其与硬件平台的紧密关联性是不可忽视的，尤其是在像昇腾这样的专用 AI 硬件架构上。为了充分发挥其计算潜能，必须采用为昇腾深度定制和优化的 Linux 内核，而非通用的内核版本。这种定制内核集成了专属的驱动程序和硬件加速模块，是上层 AI 软件栈(如 CANN)能够高效管理和使用昇腾 AI 处理器的基石。

回顾 Linux 内核的发展历程，它已从一个特定平台的项目演变为一个支持多种处理器体系结构和硬件设备的通用内核。其成功的背后离不开开源协作模式，内核源代码由官方网站 kernel.org 统一发布，允许全球开发者自由使用、研究和修改。这种开放性正是厂商能

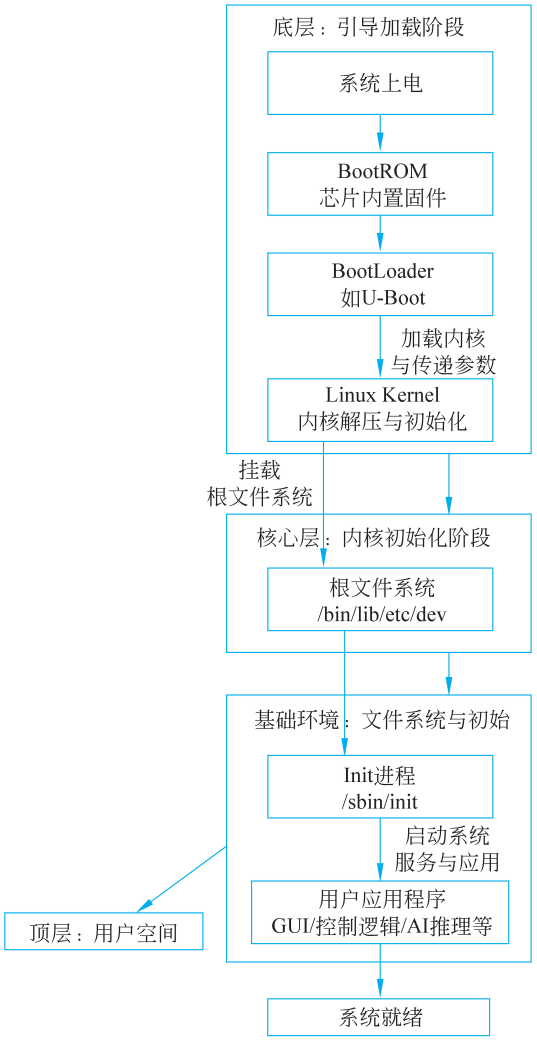

图 3.1　昇腾操作系统运行流程

够为特定硬件(如昇腾)进行二次开发的基础。在嵌入式系统和需要长期稳定运行的服务器领域,基于 Linux 社区的长期支持版本(LTS)进行开发已成为行业最佳实践。这些 LTS 版本,例如历史上重要的 4.19、5.10 以及较新的 6.1 等,会获得长达数年的安全与维护更新,为嵌入式设备、工业控制系统及云服务器提供了至关重要的稳定性和安全性保障。

对于开发者或系统管理员而言,准确识别当前运行的内核版本是一项基本技能。内核版本信息被清晰地定义在源代码顶层目录的 Makefile 文件中,通过 VERSION、PATCHLEVEL 和 SUBLEVEL 这三个变量的组合,便可以精确地确定内核的主版本号。例如,数值 6、10 和 0 共同构成了版本 6.10.0。这一方法简单而有效,是区分不同内核版本的基础。因此,在昇腾的开发和部署环境中,通过命令查询或查验 Makefile 来确认所运行的是否为官方提供的、经过验证的定制内核版本,是确保整个 AI 应用稳定和高效的第一步。

其中，昇腾 310B 内核源码包 Ascend310B-source-opi.tar.gz，解压后得到的 Linux 内核源码包，是一个为昇腾 310B 硬件平台深度定制的完整开发工作目录。其内容不仅包含标准的 Linux 内核源代码，还集成了与昇腾硬件紧密相关的驱动、设备树以及构建脚本。kernel 目录通常是整个代码包的核心，它包含了经过修改和配置的、与特定 LTS 版本同步的官方 Linux 内核源代码，其中会包含为适配昇腾芯片而加入的底层内核补丁和优化。driver 目录则至关重要，它存放了昇腾 AI 处理器的专用内核驱动模块，这些驱动负责实现操作系统与昇腾 310B 计算核心、内存及控制单元之间的通信和管理，是硬件能够被上层 AI 计算架构（如 CANN）调用的基础。

为了确保内核能够正确识别和初始化昇腾 310B 平台上的所有硬件组件，dtb 目录提供了设备树源码文件。设备树以一种数据结构的形式，精确描述了主处理器、内存、外设以及昇腾 AI 处理器等硬件资源的拓扑和配置信息，在内核启动阶段被加载，是实现跨平台支持的关键。整个内核的构建过程由一系列自动化脚本管理，其中 build.sh 作为顶层构建脚本，为用户提供了编译和打包内核的便捷入口，它内部会调用标准的 Kbuild 系统；而 scripts 和 tools 目录则包含了内核构建系统本身所需的各种辅助脚本和实用工具，用于配置、编译、链接和生成最终的内核镜像。

此外，config 目录预置了针对昇腾 310B 平台和不同应用场景的内核配置文件，在构建时选择相应的配置可以确保生成的内核启用了所有必要的功能和驱动。abl 目录可能包含了与特定 BootLoader 或平台固件相关的代码或接口定义，用于确保内核能够与引导程序顺畅对接。最后，build 目录通常是在执行构建脚本后自动生成的，用于存放所有编译过程中产生的临时文件、对象文件以及最终生成的内核镜像、驱动模块等产物，清理此目录可以视为一次彻底的“make clean”操作。

总而言之，这个源码包提供了一个开箱即用的、高度集成化的环境，使开发者能够专注于为昇腾 310B 硬件编译和生成量身定制的 Linux 内核。

在主计算机上通过 bash build.sh kernel 命令，配置 Linux 内核，如图 3.2 所示。

编译完成后，终端通常会输出如下信息：

```
generate /opt/Ascend310B-source-opi/output/kernel_modules success!
generate /opt/Ascend310B-source-opi/output/Image success!
sign /opt/Ascend310B-source-opi/output/Image success!
```

编译后的 Image 文件会存放于 Ascend310B-source-opi/output 目录下。然后执行如下命令，更新 Image 文件，也更新了内核。

```
dd if=Image of=/dev/mmcblk1 count=61440 seek=32768 bs=512
```

3.2.3　Linux 操作系统驱动开发

在 Linux 系统中，驱动程序作为硬件设备与操作系统及应用软件之间的关键桥梁，承担着初始化管理硬件资源、将硬件功能抽象为统一软件接口以及处理硬件中断事件等核心任务。面对不同的项目需求，如性能指标、开发周期和硬件复杂性等因素，开发者可以从三种主要实现方案中进行选择：完全运行在内核空间的传统驱动、完全在用户空间运行的简易

```
Libmenus --->  (or empty submenus ---). Highlighted letters are hotkeys. Pressing <Search>. Legend: [*] built-in [ ]
excluded <M> module < > module capable

    General setup --->
  2   [*] Support DMA zone
  3   [*] Support DMA32 zone
  4     Platform selection --->
  5     Enable Livepatch --->
  6     Kernel Features --->
  7     Boot options --->
  8     Power management options --->
  9     CPU Power Management --->
 10     Firmware Drivers --->
 11   [*] ACPI (Advanced Configuration and Power Interface) Support --->
 12     [*] APEI support for DTS
 13     [*] Enable DTS support APEI by default
 14   [*] Virtualization --->
 15   [*] ARM64 Accelerated Cryptographic Algorithms
 16     General architecture-dependent options --->
 17   [*] Enable loadable module support
 18   [*] Enable the block layer
 19     IO Schedulers --->
 20   [ ] The legacy percpu rw semaphores
 21     Executable file formats --->
 22     Memory Management options --->
 23   [*] Networking support --->
 24     Device Drivers --->
 25     File systems --->
 26   -- Support Memory Partitioning and Monitoring
 27     Security options --->
 28   -- Cryptographic API
 29     Library routines --->
 30     Kernel hacking --->
```

图 3.2　配置 Linux 内核

驱动，以及对硬件进行最直接控制的寄存器级驱动。这三种方案构成了一个从开发简易性到性能极致的完整技术谱系。

1. 传统内核驱动

传统内核驱动作为最经典且功能完整的实现方式，采用内核模块的形式被编译为.ko 文件，支持动态加载和卸载，并完全运行在操作系统的内核态。在现代嵌入式系统中，其技术实现主要依赖三个核心要素：设备树机制通过硬件描述文件实现驱动代码与具体硬件平台的解耦，驱动使用一系列函数解析设备树节点以获取寄存器基地址和中断号等关键资源；文件操作接口通过实现 file_operations 结构体定义设备行为，包含 open、release、read、write 和 ioctl 等函数指针，形成用户空间系统调用与驱动内部处理函数的映射关系；设备节点在/dev 目录下创建设备文件，为用户空间应用程序提供标准化的访问入口。这种驱动架构的优势在于其卓越的性能表现，运行在内核态而无须上下文切换，具有极低的访问延迟，同时能够完整利用内核功能，包括中断处理、DMA 操作和精细内存管理，并能集成到各类内核标准框架中。然而，这种方案也面临开发复杂度高、调试困难以及部署和维护复杂

等挑战，主要适用于片上系统中的专用加速器、复杂传感器融合模块、网络设备等高复杂度、高性能要求的应用场景。

2. 纯用户空间驱动

纯用户空间驱动为降低开发复杂度提供了有效解决方案，通过牺牲部分性能来获得更高的开发效率和灵活性。该方案主要基于两种技术实现路径：Sysfs 接口通过虚拟文件系统将内核中的设备信息以目录和文件形式映射到用户空间的/sys 目录下，开发者可以使用简单的文件读写命令直接控制设备，如 GPIO 控制可通过/sys/class/gpio 目录下的 export、direction 和 value 文件完成引脚导出、方向配置和电平读写；UIO 框架则通过内核中简化的通用驱动实现设备内存映射和中断捕获，用户空间程序通过 read/poll 在/dev/uioX 设备文件上等待中断事件，并通过 mmap 将设备物理寄存器映射到进程地址空间进行直接访问。这种架构的显著优势在于开发简单性，开发者无须掌握复杂的内核编程知识，可以使用熟悉的用户空间编程语言和调试工具，同时驱动错误仅导致用户进程崩溃而不会影响系统整体稳定性。但其局限性体现在性能瓶颈和功能受限两个方面，频繁的系统调用导致用户态与内核态切换开销，且难以实现复杂的 DMA 操作，主要适用于简单外设控制、快速原型验证以及对实时性要求不高的低速数据采集应用。

由于传统 Sysfs 接口在性能和设计上存在一些缺陷，Linux GPIO 子系统已经推出了更先进的替代方案，形成了新的用户空间驱动最佳实践。

（1）libgpiod：这是官方推荐并旨在取代旧 Sysfs GPIO 接口的现代 C 库和工具集。它基于 Linux 内核引入的新的 GPIO 字符设备接口，解决了 Sysfs 接口中的竞态条件、不支持中断轮询等问题。它提供了更清晰、更安全的 API（如 gpiod_chip_open、gpiod_line_request_output）和命令行工具（如 gpioset、gpioget），成为当前在用户空间操作 GPIO 的首选方案。

（2）librgpio：这是一个更现代、旨在提供友好的 API 的 GPIO 库。它也构建在新的 Linux GPIO 字符设备接口之上，体现了社区向新标准的靠拢，为开发者提供了另一种符合现代 Linux 内核设计的选择。

（3）pigpio：这是一个功能非常丰富的库，它不仅支持本地 GPIO 访问，还支持远程网络 GPIO 控制，这对于分布式应用非常有用。此外，它提供了旧 Sysfs 接口所不具备的高级功能，如硬件 PWM、波形生成和软件定时器，使其在需要精确时序控制的应用（如驱动舵机、生成特定信号）中非常受欢迎。

总结而言，纯用户空间驱动架构，特别是其 GPIO 控制部分，已经历了一次重要的现代化演进。对于新的项目，开发者应优先选择基于 GPIO 字符设备的 libgpiod，而不是已被弃用的 Sysfs 接口。而对于需要高级功能或远程控制的应用，pigpio 则是一个强大的补充选择。这些现代工具在保留用户空间驱动开发简便、调试友好等核心优势的同时，在性能、安全性和功能丰富性上都实现了显著提升，使其继续在简单外设控制、快速原型验证以及对实时性要求不高的低速数据采集应用中扮演着不可替代的角色。

3. 基于寄存器直接操作的驱动实现

基于寄存器直接操作的驱动在本质上属于传统内核驱动的特殊实现形式，通过摒弃内核现有硬件抽象层来实现极致的性能和完全的控制权。其技术实现建立在三个关键基础上：内存映射机制通过 of_iomap 或 ioremap 函数将设备寄存器的物理地址映射到内核虚拟地址空间；直接寄存器访问要求使用 ioread32、iowrite32 等专用函数进行操作，严禁普通

指针解引用，以确保正确处理具有副作用的设备寄存器；对硬件手册的依赖要求开发者必须深入理解芯片参考手册中每个控制寄存器、状态寄存器和数据寄存器的位字段定义。这种方案的突出优势在于极致性能和完全控制，消除了中间抽象层带来的开销，能够实现最直接、快速的硬件控制，支持通过标准抽象层无法达成的特殊操作。然而，它同样面临最严峻的挑战，包括最高的开发难度、较差的可移植性以及巨大的操作风险，错误的寄存器操作可能导致设备损坏或系统崩溃。因此，这种方案通常仅在对性能有极端要求的特定场景中使用，如高频金融交易系统中的网卡驱动、科学计算领域的专用硬件加速器，或是为尚未被主线内核支持的特定硬件编写驱动。

综上所述，Linux 驱动程序的三种实现方式构成了一个完整的技术选择体系。纯用户空间驱动以其卓越的简易性成为快速原型开发和简单控制的首选方案；传统内核驱动在性能、功能和集成度方面实现了最佳平衡，是大多数正规、复杂驱动开发的标准选择；而基于寄存器直接操作的驱动则作为一把锋利的双刃剑，为专家级开发者在特定场景下追求极致性能提供了技术可能。在实际项目开发中，合理的技术选型需要从开发效率、性能要求、硬件复杂性以及长期维护成本等多个维度进行综合权衡，选择最适合具体应用需求的实现路径。这种分层式的驱动架构充分体现了 Linux 系统在设计上的灵活性和适应性，能够满足从嵌入式设备到高性能服务器等不同场景的多样化需求。

3.2.4 昇腾开发板接口开发

在昇腾香橙派开发板的应用开发中，对硬件接口的访问与控制主要是通过 wiringOP 库实现。wiringOP 作为一个专门为香橙派系列开发板设计的高层硬件操作库，极大地简化了 GPIO、I^2C、SPI 等硬件外设的编程难度。该库通过封装底层复杂的寄存器操作和系统调用，为开发者提供了一套类似 Arduino 的直观 API，使得开发者无须深入理解底层硬件细节就能快速实现硬件控制功能。

1. wiringOP 库实现原理

wiringOP 库的实现原理与树莓派的 wiringPi 库高度相似，其核心在于通过系统调用和文件操作来实现对硬件资源的用户空间访问。具体来说，wiringOP 的实现建立在 Linux 内核提供的 GPIO、PWM、SPI 等子系统之上，通过操作这些子系统在/sys 和/dev 目录下暴露的设备文件来完成硬件控制。

在 GPIO 控制方面，wiringOP 主要利用 sysfs 接口实现。当开发者调用 pinMode()函数设置引脚方向时，库内部会向/sys/class/gpio/export 文件写入引脚编号，使得该引脚在 sysfs 中可见，然后向对应的/sys/class/gpio/gpioX/direction 文件写入“in”或“out”来配置输入输出模式。对于电平读写操作，digitalWrite()和 digitalRead()函数分别通过读写/sys/class/gpio/gpioX/value 文件来实现。这种基于文件系统的抽象层虽然带来了一定的性能开销，但提供了良好的安全性和隔离性。

对于 PWM 功能，wiringOP 通过 sysfs 中的 PWM 接口实现。在调用 pwmWrite()时，库函数会操作/sys/class/pwm/pwmchipX 目录下的 export、period、duty_cycle 和 enable 等文件，通过设置周期、占空比和使能参数来生成 PWM 波形。这种实现方式支持硬件 PWM 和软件 PWM 两种模式，硬件 PWM 依赖芯片的专用 PWM 控制器，而软件 PWM 则通过精度较高的定时器中断模拟实现。

在中断处理方面，wiringOP 采用轮询与内核事件通知相结合的机制。当使能引脚中断时，库内部会通过 poll()或 select()系统调用监听/sys/class/gpio/gpioX/value 文件的状态变化，一旦检测到电平跳变就触发用户注册的回调函数。这种实现虽然不能达到真正的硬件中断响应速度，但对于大多数应用场景已经足够。

此外，wiringOP 还通过操作/dev/i2c-X 和/dev/spidevX.X 等设备文件来实现 I^2C 和 SPI 通信。库函数使用 ioctl()系统调用进行设备配置和数据传输，封装了底层复杂的参数设置过程，为开发者提供了简洁的读写接口。

2. wiringOP 库应用开发

在具体实现层面，wiringOP 库在用户空间运行，以 GPIO 为例，开发者只需调用简单的 digitalWrite()、digitalRead()、pinMode()等函数即可完成引脚的初始化、电平读写等操作。这种抽象层的存在使得代码编写变得十分简洁，几行代码就能实现 LED 闪烁、按键检测等基础功能，大幅降低了嵌入式开发的门槛。

wiringOP 库的另一个重要优势在于其良好的兼容性和易用性。库函数内置了对昇腾香橙派不同型号开发板的支持，能够自动识别硬件平台并适配正确的引脚映射关系。开发者可以通过 gpio readall 命令直观地查看所有引脚的编号和功能定义，无须手动查阅复杂的芯片数据手册。测试 gpio readall 命令的输出如图 3.3 所示。

```
(base) HwHiAiUser@orangepi:~$ gpio readall
 +------+-----+----------+--------+---+  AI PRO  +---+--------+----------+-----+------+
 | GPIO | wPi |   Name   |  Mode  | V | Physical | V |  Mode  | Name     | wPi | GPIO |
 +------+-----+----------+--------+---+----++----+---+--------+----------+-----+------+
 |      |     |     3.3V |        |   |  1 || 2  |   |        | 5V       |     |      |
 |   76 |   0 |     SDA7 |    OFF | 0 |  3 || 4  |   |        | 5V       |     |      |
 |   75 |   1 |     SCL7 |    OFF | 0 |  5 || 6  |   |        | GND      |     |      |
 |  226 |   2 | GPIO7_02 |    OFF | 0 |  7 || 8  | 0 | OFF    | UTXD0    | 3   | 14   |
 |      |     |      GND |        |   |  9 || 10 | 0 | OFF    | URXD0    | 4   | 15   |
 |   82 |   5 | GPIO2_18 |    OFF | 0 | 11 || 12 | 0 | OFF    | GPIO7_03 | 6   | 227  |
 |   38 |   7 | GPIO1_06 |     IN | 1 | 13 || 14 |   |        | GND      |     |      |
 |   79 |   8 | GPIO2_15 |     IN | 1 | 15 || 16 | 1 | IN     | GPIO2_16 | 9   | 80   |
 |      |     |     3.3V |        |   | 17 || 18 | 1 | IN     | GPIO0_25 | 10  | 25   |
 |   91 |  11 | SPI0_SD0 |    OFF | 0 | 19 || 20 |   |        | GND      |     |      |
 |   92 |  12 | SPI0_SDI |    OFF | 0 | 21 || 22 | 1 | IN     | GPIO0_02 | 13  | 2    |
 |   89 |  14 | SPI0_CLK |    OFF | 0 | 23 || 24 | 0 | OFF    | SPI0_CS  | 15  | 90   |
 |      |     |      GND |        |   | 25 || 26 | 1 | IN     | GPIO2_19 | 16  | 83   |
 |      |     |     SDA6 |        |   | 27 || 28 |   |        | SCL6     |     |      |
 |  231 |  17 |    URXD7 |    OFF | 0 | 29 || 30 |   |        | GND      |     |      |
 |   84 |  18 | GPIO2_20 |     IN | 1 | 31 || 32 | 1 | IN     | PWM3     | 19  | 33   |
 |  128 |  20 | GPIO4_00 |     IN | 1 | 33 || 34 |   |        | GND      |     |      |
 |  228 |  21 | GPIO7_04 |    OFF | 0 | 35 || 36 | 0 | OFF    | GPIO2_17 | 22  | 81   |
 |    3 |  23 | GPIO0_03 |     IN | 1 | 37 || 38 | 1 | IN     | GPIO7_06 | 24  | 230  |
 |      |     |      GND |        |   | 39 || 40 | 0 | OFF    | GPIO7_05 | 25  | 229  |
 +------+-----+----------+--------+---+----++----+---+--------+----------+-----+------+
 | GPIO | wPi |   Name   |  Mode  | V | Physical | V |  Mode  | Name     | wPi | GPIO |
 +------+-----+----------+--------+---+  AI PRO  +---+--------+----------+-----+------+
```

图 3.3　wiringOP 库测试 gpio readall 命令输出

此外，该库还提供了软件 PWM、延时控制、中断处理等高级功能，能够满足大多数常见嵌入式应用场景的需求。

结合图 3.3 以 7 号引脚（对应 GPIO 为 GPIO7_02、对应 wPi 序号为 2）为例，演示如何设置 GPIO 口的方向和高低电平。

（1）首先设置 GPIO 口为输出模式，其中第三个参数需要输入引脚对应的 wPi 的序号。

```
(base) HwHiAiUser@orangepi:~$ gpio mode 2 out
```

(2) 然后使用以下命令可以查看 GPIO 当前的模式,可以看到当前的模式为输出(OUT)。命令中第二个参数需要输入引脚对应的 wPi 序号。

```
(base) HwHiAiUser@orangepi:~$ gpio qmode 2 OUT
```

(3) 然后就可以开始设置引脚输出高低电平了。例如,使用以下命令可以设置 GPIO 口输出低电平,设置完成后可以使用万用表测量引脚的电压值,如果为 0V,说明设置低电平成功。

```
(base) HwHiAiUser@orangepi:~$ gpio write 2 0
```

(4) 除了使用万用表测量引脚的电压外,还可以使用 gpio read 命令查看引脚的高低电平状态。当前命令输出为 0,说明引脚目前为低电平。

```
(base) HwHiAiUser@orangepi:~$ gpio read 2 0
```

(5) 然后可以设置 GPIO 口输出高电平,设置完成后可以使用万用表测量引脚的电压值,如果为 3.3V,说明设置高电平成功。

```
(base) HwHiAiUser@orangepi:~$ gpio write 2 1
```

(6) 除了使用万用表测量引脚的电压外,还可以使用 gpio read 命令查看引脚的高低电平状态。当前命令输出为 1,说明引脚目前为高电平。

```
(base) HwHiAiUser@orangepi:~$ gpio read 2 1
```

同时也可以通过 digitalWrite()、digitalRead()、pinMode()等函数在程序中控制外部硬件,同样针对 GPIO7_02,可以利用如下语句来实现对 IO 口的设置。

```
pinMode(2, OUTPUT);                    //设置 GPIO7_02(即 2 号引脚)为输出
digitalWrite(2, HIGH);                 //设置 GPIO7_02(即 2 号引脚)高电平
digitalRead(2);                        //读取 GPIO7_02(即 2 号引脚)
```

然而,这种便利性也带来了一定的性能代价。由于 wiringOP 运行在用户空间,每次硬件操作都需要经过用户态到内核态的上下文切换,这会引入额外的延迟和 CPU 开销。对于需要精确时序控制或高实时性的应用场景,这种延迟可能成为系统瓶颈。同时,库的抽象层也限制了开发者对硬件的精细控制能力,无法实现某些特殊的底层操作。

总体而言,wiringOP 库为昇腾香橙派开发板提供了一种平衡了易用性和功能性的硬件访问方案。它特别适合快速原型开发、教学演示和中等复杂度的嵌入式项目,能够帮助开发者专注于应用逻辑的实现,而不必纠结于底层硬件细节。对于大多数应用场景来说,wiringOP 提供的性能和控制能力已经足够,这也是它成为昇腾香橙派生态中最为流行的硬件编程库的主要原因。

3.2.5 昇腾 NPU-SMI 工具

NPU-SMI 是昇腾 AI 处理器生态中至关重要的系统管理工具,其功能定位与英伟达的

NVIDIA-SMI 类似，专门用于监控、管理和维护搭载昇腾 AI 芯片的硬件设备。该工具通过命令行界面为用户和系统管理员提供了对 NPU 硬件状态的全面洞察和基础控制能力，是昇腾 AI 应用开发、部署和运维过程中不可或缺的组成部分。

在信息监控方面，NPU-SMI 能够提供详尽的硬件状态数据。启动该工具后，用户首先看到的是系统中所有昇腾 AI 处理器的整体概览，包括设备数量、物理位置、芯片型号、固件版本等基础信息。进一步深入，它可以实时显示每个 NPU 芯片的核心健康指标：温度传感器读数，确保芯片工作在安全的热设计范围内；功耗监控单元提供实时功率消耗数据，帮助用户优化能效比；内存控制器则报告 HBM 或 DDR 内存的使用情况，包括总容量、已使用量和剩余可用量。此外，工具还提供算力利用率指标，直观展示 AI 计算单元的实际负载水平，这些数据对于性能调优和瓶颈分析具有重要价值。

在设备管理功能层面，NPU-SMI 支持对异常设备执行必要的控制操作。当某个 NPU 设备出现异常或需要重新初始化时，管理员可以通过该工具执行硬件复位操作，这将清除设备当前状态并恢复至初始就绪状态。在功耗管理方面，该工具支持多种电源模式的切换，用户可以根据实际应用场景在高性能模式和节能模式之间进行选择，实现计算性能与能耗的平衡。对于需要精确控制运行环境的场景，还可以设置温度阈值，当芯片温度超过预设值时自动触发保护机制。

在任务管理维度，NPU-SMI 具备完善的进程监控功能。它可以列出当前正在使用 NPU 计算资源的所有运行中进程，详细显示每个进程的 PID、所属用户、占用 NPU 内存大小、计算核心使用情况等信息。当出现进程异常或需要释放资源时，管理员可以直接通过工具终止指定进程，这种能力在共享开发环境或生产服务器中尤为重要，能够有效防止资源被异常进程长期占用。

从系统维护角度，NPU-SMI 集成了强大的日志收集和诊断功能。它可以一键获取与 NPU 相关的所有系统日志、驱动日志和运行日志，这些日志信息对于分析硬件故障、驱动兼容性问题或应用程序错误具有关键作用。同时，工具还提供硬件自检功能，能够快速诊断 NPU 设备的基本健康状态，包括内存测试、计算核心测试等，确保硬件在投入使用时处于正常状态。

在实际使用中，NPU-SMI 既支持单次查询模式，也提供实时监控模式。用户可以通过简单的命令行参数获取特定信息，也可以进入交互式界面持续观察系统状态变化。这种灵活性使得它既适用于快速的系统检查，也适合长期的运行监控。无论是开发者在调试模型时确认硬件资源占用情况，还是运维人员在生产环境中监控系统健康状态，NPU-SMI 都能提供必要的信息支持。

总体来说，NPU-SMI 作为昇腾 AI 处理器的标准管理工具，建立了一个完整的硬件管理生态系统。它将复杂的底层硬件信息封装成易于理解的指标，将危险的低级操作转化为安全的管理命令，极大地降低了昇腾 AI 平台的使用和维护门槛。随着昇腾 AI 生态的不断发展，NPU-SMI 的功能也在持续丰富和完善，为构建稳定可靠的 AI 计算基础设施提供了坚实保障。

例如，开发板使用的昇腾 SoC 总共有 4 个 CPU，这 4 个 CPU 既可以设置为 control CPU，也可以设置为 AI CPU。默认情况下，control CPU 和 AI CPU 的分配数量为 3∶1。

使用 npu-smi info 命令可以查看 control CPU 和 AI CPU 的分配数量。

```
(base) HwHiAiUser@orangepiaipro:~$ npu-smi info -t cpu-num-cfg -i 0 -c 0
    Current AI CPU number : 1
    Current control CPU number : 3
    Current data CPU number : 0
```

当 Linux 系统跑满后，使用 htop 命令会看到有一个 CPU 的占用率始终接近 0，请注意，这是正常的。因为这个 CPU 默认用于 AI CPU。

如果当前环境模型中无 AI CPU 算子，且运行业务时查询 AI CPU 占用率持续为 0，则可以将 AI CPU 的数量配置为 0。查询 AI CPU 占用率的命令如下所示：

```
(base) HwHiAiUser@orangepiaipro:~$ npu-smi info -t usages -i 0 -c 0
    Memory Capacity(MB) : 7545
    Memory Usage Rate(%) : 20
    Hugepages Total(page) : 15
    Hugepages Usage Rate(%) : 100
    Aicore Usage Rate(%) : 0
    Aicpu Usage Rate(%) : 0
    Ctrlcpu Usage Rate(%) : 1
    Memory Bandwidth Usage Rate(%) : 1
```

如果不需要使用 AI CPU，可使用下面的命令将 4 个 CPU 都设置为 control CPU。设置完成后需要重启系统，使配置生效。

```
(base) HwHiAiUser@orangepiaipro:~$ sudo npu-smi set -t cpu-num-cfg -i 0 -c 0 -v
0:4:0
Status : OK
Message : The cpu-num-cfg of the chip is set successfully. Reset system for the
configuration to take effect.
```

当 4 个 CPU 都设置为 control CPU 后，再运行任务让所有 CPU 跑满，使用 htop 命令就能看到 4 个 CPU 的占用率都能达到 100%了。

3.2.6 Linux 系统构建方法

在嵌入式 Linux 开发领域，系统构建方法呈现出多元化的技术路线，为开发者提供了不同层次的解决方案。从高度自动化的构建框架到完全手动的脚本编写，每种方法都体现了独特的工程理念和适用场景。

Buildroot 作为轻量级构建方案的代表，以其简洁的 Kconfig 配置系统和高效的构建流程著称。它能够快速集成交叉编译工具链、Linux 内核和各种软件包，生成精简的固件镜像。这种“开箱即用”的特性，使其成为中小型项目和快速原型开发的理想选择，特别适合那些对系统体积和构建速度有严格要求的应用场景。

Yocto 项目则面向更复杂的工业级应用需求，提供了一套完整的构建框架和强大的元数据架构。通过 BitBake 构建引擎和分层机制，Yocto 支持高度定制化的系统配置和软件包管理。其突出的优势在于能够创建可重复、可维护的构建环境，并支持运行时软件包管理和在线更新功能，因此在汽车电子、工业自动化和物联网基础设施等领域得到广泛应用。

对于追求极致控制和深度定制的开发者而言，手动编写构建脚本提供了最高程度的灵活性。这种方法要求开发者从工具链搭建开始，逐步实现内核配置、软件包编译和根文件系统构建的每一个环节。虽然这种方法需要深厚的系统知识和大量的开发投入，但它能够实现最精细的系统优化和最小化的资源占用。

这些构建方法在易用性、灵活性和功能性方面形成了明显的权衡关系。Buildroot 在易用性和构建效率方面表现突出，Yocto 在系统可维护性和可扩展性方面具有优势，而手动构建则在系统优化和深度控制方面无可替代。开发者需要根据项目的具体需求、团队的技术储备和产品的生命周期要求，选择最适合的构建方案。

当前，这些构建方法在嵌入式领域形成了互补共生的生态格局，各自服务于不同的应用场景和开发阶段。从快速原型开发到大规模产品部署，从资源受限的嵌入式设备到功能复杂的边缘计算节点，多元化的构建方案为嵌入式 Linux 的广泛应用提供了坚实的技术基础，持续推动着整个行业的技术创新和发展。

昇腾香橙派开发板采用专门定制的系统构建框架，分别提供了最小镜像(Minimal)、完整镜像(Complete)、压缩扩容镜像(Compress)三个模块版本，具体功能如表 3.1 所示。

表 3.1　昇腾香橙派开发板系统镜像模块功能说明

模 块 名 称	功 能 简 介
最小镜像(Minimal)	可以在开发板上启动但缺少部分依赖的镜像
完整镜像(Complete)	完整依赖镜像
压缩扩容镜像(Compress)	带有压缩扩容功能的完整依赖镜像

以最小镜像的制作过程展开描述，包括如下文件，如图 3.4 所示。

图 3.4　制作最小镜像的文件

文件中包括针对 Ubuntu 和 openEuler 的脚本及其他文件，分别在 Ubuntu 和 openEuler 文件夹中，其中在 Ubuntu 的 download 目录下有针对昇腾的文件和配置文件以及 Ubuntu 镜像，如图 3.5 所示。

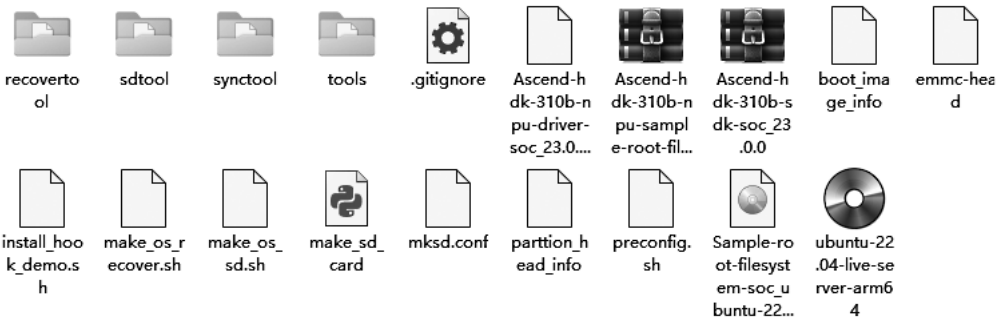

图 3.5　针对昇腾的文件和配置文件以及 Ubuntu 镜像

图 3.4 中的 base.sh 脚本是一个用于为昇腾 AI 处理器开发板构建和打包定制 Linux 系统镜像的自动化工具。其核心设计理念是模块化与可配置性。脚本本身作为一个“框架”或“引擎”，并不直接包含构建系统的具体步骤（如分区、安装软件包等），而是通过解析外部的配置文件和一个独立的函数库，动态地决定和执行构建流程。这种设计将“做什么”（在配置文件中定义）与“怎么做”（在函数库中实现）分离开，使得脚本极具灵活性，能够通过修改配置来适配不同的 Linux 发行版、版本号和功能需求，而无须改动脚本核心逻辑。

（1）参数解析与用户引导。

脚本首先处理用户输入的命令行参数，这是用户与脚本交互的入口。它定义了清晰的用法说明（Usage），告知用户必需的参数（发行版路径和目标磁盘设备）和可选参数（依赖下载路径和后处理挂载路径）。同时，它提供了 -h 选项来显示详细的帮助信息，以及 -v 选项来开启详细输出模式。在详细模式下，脚本执行过程中所有命令的原始输出都会显示给用户，这对于调试和了解构建细节至关重要。而默认的静默模式则只显示脚本自定义的日志信息，使输出更加清晰简洁。这种设计兼顾了普通用户和开发者的需求。

（2）环境初始化与变量设定。

在解析完参数后，脚本进入环境初始化阶段。它通过一系列环境变量的设定，为后续所有操作构建上下文。这些变量包括脚本自身的路径、关键的配置文件和函数库文件的名称（如 cfg.json 和 func.sh），以及从用户输入中推导出的关键信息，如发行版名称（例如 Ubuntu）、版本号（例如 22.04）和目标磁盘（例如/dev/sdb）。此外，脚本还为可选的依赖下载路径和后处理挂载路径设置了默认值，确保即使用户未指定，脚本也能在预定的目录结构下正常工作。这个阶段为整个构建过程奠定了坚实的基础和数据依据。

（3）核心支撑功能模块。

为了确保脚本的健壮性和可维护性，它内置了一套强大的核心功能函数库。日志函数为所有操作提供了带时间戳的统一记录，是跟踪脚本进度的主要信息来源。命令执行辅助函数是脚本安全性的关键，它封装了所有系统命令的调用，能够自动捕获命令执行失败的情况。一旦任何命令执行出错，该函数会立即触发异常处理流程，并打印出失败的命令，然后终止脚本，防止在错误的状态下继续执行。依赖下载函数体现了脚本的智能化，它首先在本地指定的缓存目录中查找所需的依赖文件（如驱动、软件包），如果找不到，则根据 cfg.json 配置文件中的 URL 自动下载，并具有简单的版本管理和备份机制。最后，错误与中断处理函数确保了脚本在任何异常退出（包括用户手动按下 Ctrl＋C 键）时，都能尽可能地执行清理工作，释放资源（如卸载已挂载的目录），避免残留中间状态污染系统。

（4）系统准备与前置检查。

在正式执行构建任务之前，脚本进行了一系列严格的检查和准备工作。权限检查确认了当前用户是否具有 root 权限，因为对磁盘进行分区、格式化以及安装系统软件都需要最高权限。参数检查则验证了用户输入的路径和设备名是否有效，例如确保目标磁盘真实存在，防止误操作导致数据丢失。同时，它会创建必要的目录结构，如下载目录和挂载点目录。系统依赖安装环节负责安装脚本自身运行以及后续构建步骤所必需的系统软件包，例如用于处理 JSON 配置文件的 jq、用于在 x86 主机上运行 ARM64 程序的 qemu-user-static，以及进行磁盘分区管理的 parted 和 kpartx 等工具。这个阶段确保了构建环境是完整、正确且可用的。

(5) 动态工作流引擎：配置驱动执行。

这是整个脚本最核心、最精巧的部分。配置处理函数扮演了“解析器”的角色，它读取并解析与特定发行版和版本对应的 cfg.json 文件。这个 JSON 文件定义了一个功能列表(例如 partition_disk、install_packages、install_cann 等)，每个功能都有一个开关(“y”或“n”)。脚本会将所有标记为“y”的功能名提取出来，形成一个待执行的“任务清单”。随后，运行控制函数扮演了“执行器”的角色。它动态地加载(source)包含具体实现细节的 func.sh 函数库，然后按照“任务清单”依次调用每个函数。每成功执行一个函数，脚本就会在配置文件中将该功能标记为“n”，这是一种简单的状态持久化机制，如果脚本中途失败，下次运行时可以跳过已完成的步骤。当所有功能都成功执行完毕后，脚本会将配置文件中的所有功能标记重置为“y”，以便下次构建可以重新开始。这种基于配置的动态执行模式，使得构建流程完全由数据(配置文件)驱动，极大地增强了其适应性和可扩展性。

(6) 执行流程与总结。

脚本的主函数将上述所有模块串联起来，形成一个完整的执行流水线。它首先设置中断信号捕获，确保用户能安全地终止脚本。然后，它严格按照顺序调用：权限检查、参数检查、依赖安装、配置解析、功能执行。这个流程设计确保了每一步都建立在前一步成功完成的基础上。最终，无论构建成功与否，脚本都会给出明确的状态信息(“Minimal image build successful!” 或 “Minimal image build failed!”)，并确保工作目录被恢复到用户最初调用脚本的位置，体现了良好的用户体验和资源管理。

总而言之，这个脚本是一个设计精良、功能强大的系统镜像构建自动化框架，它通过高度的模块化和配置化，专门为简化昇腾开发板的系统定制与部署流程而生。

3.3　昇腾应用软件开发

3.3.1　应用程序开发流程

在昇腾香橙派和 Atlas 200I DK 开发板的 Linux 操作系统中，开发一个最简单的 Hello World 程序的过程类似，因为它们都运行基于 ARM64 架构的 Linux 系统。

1. 环境准备与确认

在开始编程之前，首先需要确保开发板系统已就绪。通过串口或 SSH 登录到开发板的 Linux 系统后，您应处于一个命令行终端界面。接下来的第一步是确认您的开发环境。

(1) 检查系统架构。在终端中输入 uname -m 并按回车键。对于这两款开发板，预期的输出都是 aarch64，这确认了您正在一个 ARM64 架构的系统上进行原生编译和开发。

(2) 检查编译器。Hello World 程序通常用 C 语言编写，并使用 GCC 进行编译。输入命令 gcc --version 来检查 GCC 编译器是否已安装。如果系统已预装，它会返回编译器的版本信息(如 aarch64-linux-gnu-gcc 等)。如果系统提示“command not found”，您需要安装编译工具链，使用命令 sudo apt update && sudo apt install gcc make 即可完成安装。

2. 创建项目与编写代码

(1) 在合适的位置(如用户主目录)，执行以下命令：

```
mkdir hello_world
cd hello_world
```

该命令创建了一个名为 hello_world 的文件夹并进入其中。

(2) 编写源代码。接下来,需要使用文本编辑器创建一个 C 语言源文件。这里以 vim 为例。

```
vim hello.c
```

在编辑器中,输入以下经典的 C 语言代码:

```
#include <stdio.h>
int main() {
    printf("Hello, World from Ascend Developer Kit!\n");
    return 0;
}
```

这段代码的功能是,包含标准输入输出头文件,定义一个主函数,并在该函数中调用 printf 向控制台输出一条问候信息,最后返回 0 表示程序正常结束。输入完成后,保存并退出编辑器(在 vim 中,按 Esc 键后输入 wq 再按回车键)。

3. 编译源代码与生成可执行文件

源代码是人类可读的文本,需要被“翻译”成机器可以执行的二进制格式。

(1) 执行编译。在终端中,使用 GCC 编译器将 hello.c 编译成可执行文件。命令如下:

```
gcc -o hello hello.c
```

其中:

- gcc:调用 GCC 编译器。
- -o hello:指定输出的可执行文件名为 hello。
- hello.c:指定输入的源代码文件。

(2) 验证编译结果。如果命令执行后没有报错并返回到命令提示符,说明编译成功。您可以通过 ls 命令看到当前目录下生成了一个名为 hello 的新文件。您可以使用 file 命令进一步验证其格式:

```
file hello
```

输出应类似于 hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), …,这确认了它是一个针对 ARM64 架构(aarch64)的可执行文件。

4. 运行程序与查看结果

(1) 运行程序。在 Linux 中,当前目录通常不在系统的可执行文件搜索路径($PATH)中,因此需要在程序名前加上 ./ 来告诉系统在当前目录查找该文件。

```
./hello
```

(2) 查看输出。按下回车键后,将在终端中看到程序的输出:

```
Hello, World from Ascend Developer Kit!
```

至此,已经成功在昇腾开发板上完成了第一个程序的开发、编译和运行的完整流程。

对于单个文件,直接使用 GCC 命令很简单。但当项目稍复杂时,使用 Makefile 管理编译过程是更专业的做法。

创建 Makefile:

```
vim Makefile
```

输入以下内容:

```
makefile
#定义编译器
CC=gcc
#定义编译选项
CFLAGS=-Wall
#定义目标可执行文件名称
TARGET=hello
#定义源文件
SRC=hello.c
#默认目标:生成可执行文件
$(TARGET): $(SRC)
$(CC) $(CFLAGS) -o $(TARGET) $(SRC)
#清理目标:删除编译生成的文件
clean:
rm -f $(TARGET)
#声明"clean"为一个伪目标,即使存在名为"clean"的文件也执行命令
.PHONY: clean
```

在使用 Makefile 时,编译程序只需输入 make;清理生成的可执行文件,输入 make clean。

3.3.2　远程应用程序开发

在人工智能与边缘计算开发领域,一种非常普遍且高效的工作模式是将计算能力强大的专用硬件(如昇腾开发板)作为远程计算节点,而开发者则使用熟悉的通用计算机(如笔记本计算机或台式机)作为主要工作界面。这种架构的核心思想是“职责分离”:主机负责提供友好、高效的代码编辑与界面交互体验,而昇腾开发板则专职负责执行需要巨大算力的 AI 模型的推理与训练任务。两者之间通过网络连接,形成一个协同工作的整体。这种模式不仅避免了在资源受限的开发板上安装笨重的 IDE 的麻烦,也使得团队协作和环境管理变得更为统一和便捷。

实现这一模式的第一步,也是基础环节,是通过安全外壳协议(Secure Shell,SSH)对昇腾开发板进行远程控制。开发者需要在主机上打开一个终端,使用一条简单的命令(例如 ssh huawei@192.168.1.100)即可建立一条加密的网络通道,连接到远程开发板。成功登录

后，这个终端窗口就完全等同于直接连接到开发板的显示器与键盘。通过这个命令行界面，开发者可以执行所有必要的系统级操作：这包括管理文件系统（创建、移动、删除项目文件）、安装和配置软件依赖（如 Python 库、CANN 工具包）、设置环境变量（指向昇腾 AI 处理器的运行时库），以及最终启动和运行 AI 应用程序。此外，SSH 连接还用于执行关键的监控命令，如使用 NPU-SMI 工具实时查看 NPU 芯片的利用率、内存占用、温度与功耗等运行指标，确保计算任务稳定执行。

然而，纯命令行界面对于复杂的代码编写和调试工作来说并不友好。为了将主机的开发体验无缝延展到远程硬件上，微软 Visual Studio Code（VS Code）编辑器及其强大的 Remote-SSH 扩展成为不可或缺的核心工具。安装此扩展后，开发者不再仅仅是建立一个命令行的连接，而是将整个 VS Code 编辑器界面“投射”到远程开发板上。其底层原理是，VS Code 会在后台通过 SSH 连接，自动在昇腾开发板上部署一个轻量级的服务器端进程。这个进程负责处理代码编辑、文件管理和扩展功能等请求，而主机上的 VS Code 界面则作为一个高效的客户端进行交互。

这种方式带来了革命性的体验提升。开发者可以在自己熟悉的主机 VS Code 环境中，直接打开、浏览和编辑存储在昇腾开发板上的整个项目代码文件。更重要的是，他们可以继续使用所有惯用的强大功能，例如智能语法高亮、自动代码补全、集成 Git 版本控制以及图形化的调试器。当需要编译和运行代码时，开发者可以直接在 VS Code 内部集成的终端里输入命令，这个终端本质上就是一个 SSH 会话，所有命令都在开发板的原生环境中执行。

将 SSH 远程控制与 VS Code 远程编辑相结合，便构成了一个流畅且强大的端到端开发工作流。开发者首先通过 VS Code 的 Remote-SSH 功能连接到昇腾开发板，在一个直观的图形化界面中编写和修改代码。随后，在编辑器内一键打开终端，直接使用昇腾工具链（如 AscendCL）进行程序的编译与运行。整个开发、调试和测试的循环都在这个集成的环境中完成。如果需要进行底层的系统诊断或性能分析，开发者则可以另外开启一个独立的 SSH 终端会话，专门用于运行监控命令。

3.3.3 应用程序开发管理

在现代软件开发中，特别是涉及团队协作的智能系统开发项目中，应用程序代码管理是确保项目有序、高效和可追溯的基石。代码管理工具集成了七大核心功能，共同构建了一个安全的开发环境：它们能完整记录项目生命周期，实现对所有变更的精确追踪，通过版本控制清晰管理不同版本的功能异同，利用权限控制保障代码安全，通过责任追究明确开发职责，提供回退处理能力以快速从错误中恢复，并内置冲突解决机制来高效处理多人协作中的代码冲突。

在众多的代码管理工具中，以 Git 为代表的分布式版本控制系统已成为业界标准。Git 本身是一个高效、灵活的核心工具，而基于它构建的在线平台则形成了丰富的生态系统。在这个生态中，主要包含三个关键组成部分。

Git：作为底层引擎，它是一个开源的分布式版本控制软件，负责处理代码的存储、版本历史和分支合并等核心操作。

GitHub：这是一个基于 Git 的全球性开源项目托管平台和开发者社区。它的独特优势在于其社交化和开放协作模式，通过“Fork”和“Pull Request”机制，极大地促进了开源项目

的代码共享和贡献。

GitLab：与 GitHub 不同，GitLab 是一个基于 Git 实现的、功能更为全面的 DevOps 平台。它不仅可以作为代码仓库，更集成了项目管理、持续集成/持续交付(CI/CD)、代码审查、容器镜像库等一系列自动化工具。其核心优势在于完善的权限管理和支持私有化部署，允许企业或机构在内网环境中搭建私有的 Git 服务，从而更好地满足代码私有性、安全性和定制化的需求，是企业级开发的优选。

此外，针对国内网络环境，Gitee(码云)提供了一个优秀的本土化替代方案。它在提供与 GitHub 类似的基础代码托管服务之外，还集成了代码质量检测、项目演示等功能，并对国内团队协作提供了良好的支持。

综上所述，从代码的私有性、安全性和 DevOps 流程自动化的深度集成角度来看，GitLab 通常被认为是企业或学校内部搭建私有代码管理平台的一个更强大、更全面的选择。开发者可以根据项目的开放性、团队规模和具体流程需求，灵活选择 GitHub、GitLab 或 Gitee 来承载其代码管理功能，并与 Git 核心工具配合使用，共同保障开发工作的顺利进行。

以一个基于 Git 的手写数字识别项目来说明代码管理流程。在该项目的开发中，使用 Git 和 GitHub 进行代码版本控制。整个过程始于项目初始化。

1. 个人项目代码维护

对于个人开发者而言，使用 Git 的主要目的是对代码版本进行系统性的维护，记录每一个关键节点，并为自己提供一份可靠的“后悔药”。

(1) 项目初始化与首次提交。

项目开始时，开发者在项目根目录下执行 git init 命令，初始化一个本地 Git 仓库。接着，创建项目的基本结构：用于模型训练的 train.py、模型推理的 inference.py 和配置文件 config.yaml。

完成基础代码编写后，通过 git add 命令将所有文件添加到 Git 的暂存区。随后使用 git commit -m "feat：项目初始化，添加基础训练和推理脚本"命令进行了第一次提交。这个提交信息清晰地记录了这一刻完成了什么工作，为项目建立了第一个坚实的基石。

(2) 日常开发与功能迭代。

在开发过程中，开发者遵循“小步快走”的提交策略。每完成一个小的、完整的功能点，就进行一次提交。

例如，当为数据预处理添加了归一化功能后，需要执行：

```
git add train.py                                          #将修改的文件加入暂存区
git commit -m "feat: 在数据预处理管道中添加像素归一化"      #提交更改
```

当修复了一个导致内存泄漏的缺陷(Bug)时，提交信息可能是：

```
git commit -m "fix: 修复推理过程中的张量内存未释放问题"
```

这种清晰的信息让开发者在未来回顾历史时，能够一目了然地知道每次修改的意图。

(3) 版本标签与里程碑管理。

当项目取得重大进展时——比如模型准确率首次达到 98%，开发者会为这个值得纪念的版本创建一个标签。

首先，使用 git log --oneline 查看提交历史，找到对应最终提交的简短 ID。然后执行 git tag -a v1.0.0 -m "模型准确率达标首版"创建一个附注标签。通过 git tag 命令，可以列出项目中创建的所有标签，清晰地看到项目发展的各个里程碑。

(4) 代码回溯与错误修复。

这是 Git 为个人开发者提供的核心价值。当某次代码修改引入了严重错误，导致训练过程崩溃时，开发者不需要手动撤销代码。

首先，使用 git status 查看当前工作区的修改情况。如果只是想放弃某些文件的临时修改，可以直接执行 git checkout -- [文件名] 来还原到最近一次提交的版本。如果需要更彻底的回归，可以使用 git log 找到最后一个稳定版本的提交 ID。然后通过 git reset --hard [commit_id] 命令，将整个项目代码库强硬地回退到那个已知的、稳定的时间点。这个操作让开发者可以毫无顾虑地进行各种代码实验，因为知道任何时候都能安全地回到过去。

(5) 分支的灵活运用。

即使是个人开发，分支也是一个非常有用的工具。当尝试一个激进的重构(比如将整个模型从 CNN 改为 Transformer)而又不想影响主线开发时，可以输入以下命令：

```
git checkout -b experiment/transformer-arch  #创建并切换到实验分支
```

在这个分支上，开发者可以自由地修改、提交。如果实验成功，可以将其合并回主分支；如果失败，简单地删除这个分支即可，主分支的代码完好无损。

通过以上这些简洁的 Git 命令和策略，可以为手写数字识别项目构建了一套完整的代码演进档案。它不仅记录了开发者是如何一步步构建出这个项目的，更重要的是，它赋予了开发者大胆重构和试验的信心，因为所有的“过去”都被妥善地保存着，随时可以重返。

2. 团队项目代码维护

(1) 项目初始化与代码托管。

项目经理首先在本地创建项目目录，并使用 git init 命令将其初始化为一个 Git 仓库。接着，在 GitHub 上创建一个名为 handwritten-digit-recognition 的远程仓库。然后，通过 git remote add origin <GitHub 仓库 URL>命令将本地仓库与远程仓库关联起来。最后，执行 git push -u origin main，将包含基础项目结构(如 model_training/、inference_engine/ 等目录)的初始代码推送到 GitHub，作为项目开发的基石。

(2) 并行开发与功能分支。

项目采用功能分支工作流来支持并行开发。当一名工程师需要开发一个新的卷积神经网络模型时，他不会直接在主干上修改，而是使用 git checkout -b feature/new-cnn-model 命令创建一个专属的功能分支。在这个独立的分支上，他可以安全地修改 model.py 和 train.py 等文件，并通过 git commit -m "实现新的 CNN 层结构"命令频繁地提交进度。这些提交在分支上形成了一条清晰的开发轨迹。

(3) 代码合并与团队协作。

当新模型功能开发完成并经过初步测试后，就需要将其集成到主项目中。工程师首先将功能分支推送到远程仓库：git push origin feature/new-cnn-model。随后，在 GitHub 界面上发起一个拉取请求(Pull Request，PR)，请求将分支合并到 main 分支。这相当于正式